

Local Lexing^{*}

Steven Obua

University of Edinburgh
steven.obua@gmail.com

Phil Scott

University of Edinburgh
phil.scott@ed.ac.uk

Jacques Fleuriot

University of Edinburgh
jdf@inf.ed.ac.uk

Abstract

We introduce a novel parsing concept called *local lexing*. It integrates the classically separated stages of lexing and parsing by allowing lexing to be dependent upon the parsing progress and by providing a simple mechanism for constraining lexical ambiguity. This makes it possible for language design to be composable not only at the level of context-free grammars, but also at the lexical level. It also makes it possible to include lightweight error-handling directly as part of the language specification instead of leaving it up to the implementation.

We present a high-level algorithm for local lexing, which is an extension of Earley's algorithm. We have formally verified the correctness of our algorithm with respect to its local lexing semantics in Isabelle/HOL.

1. Introduction

The traditional approach to specifying the syntax of a computer language is to define two components, a *lexer* (also called *scanner*) and a *parser*. The lexer partitions the input document consisting of a sequence of *characters* into a sequence of *tokens*. Each token is uniquely associated with a *terminal*, such that the sequence of tokens can be viewed as a sequence of terminals. The parser is typically defined by a *context-free grammar* (CFG), and checks if the sequence of terminals is in the language generated by the CFG.

CFGs are a powerful language design tool. A most important property of CFGs is composability. Given two or more CFGs, it is easy to combine them into a single CFG in various ways in order to specify a language as a mashup of several other languages, which is a common scenario in modern programming environments.

The lexer component of this traditional setup is problematic though in such a mashup scenario. A keyword in one language might be an identifier in another language, rendering the lexers of these two languages incompatible with each other. The problem is that lexers are not supposed to be composed in the traditional setup. Practical solutions to this problem usually involve some form of ad-hoc communication between parser and lexer stages, thus shifting an issue

which should be dealt with at the language design level to the implementation level.

The main reasons for the traditional split of syntax recognition into lexing and parsing are *speed*, *expressivity* and *convenience*:

Speed Typically terminals are specified via regular expressions that can be recognized with deterministic finite state machines. This is usually much faster than parsing with respect to a CFG.

Expressivity Despite CFGs being more expressive than regular expressions, two common lexing rules cannot be expressed with CFGs: *longest-match*, and *priority*. The longest-match rule states that if multiple terminals are associated with regular expressions that could all match prefixes of the rest of the sequence of characters to be scanned, then the terminals which match the longest prefix are to be preferred. The priority rule comes into play if after application of the longest-match rule there are still at least two different terminals left as possible candidates: then a linear priority order among terminals is assumed, and the terminal with the highest priority among all candidates is picked.

Convenience In this setup, whitespace and comments are usually special terminals which do not appear in any grammar rule, but which are filtered out of the sequence of terminals during the lexing stage.

Scannerless parsing [3] proposes to solve our lexing problem by relinquishing separate lexing, effectively identifying characters, tokens and terminals. This negatively affects all three mentioned advantages of a separate lexing stage, the most severely affected being expressivity: in order to approximate the missing longest-match and priority rules, scannerless parsing introduces follow restrictions and reject productions, which are awkward additions to the elegant formalism of context-free grammars because they destroy the nice composability properties of context-free grammars. A scannerless parsing technique called *packrat parsing* goes even further and does away with context-free grammars entirely, employing *parsing expression grammars* instead [2], again to the detriment of composability.

Instead, we propose a new parsing semantics which we call *local lexing*. Local lexing keeps the distinction between

^{*} This work has been funded by EPSRC grant [EP/L011794/1](#).

lexing and parsing, and between characters, tokens and terminals. But instead of deterministically converting a sequence of characters into a sequence of tokens, local lexing converts a sequence of characters into a *set* of token sequences, applying lexing and parsing in an intertwined manner.

The contributions of this paper are as follows: We first define the novel concept of local lexing in Section 2. We then present examples of applications of local lexing in Section 3. These examples show that local lexing is a generalisation of the traditional setup, and that local lexing readily allows integrated access at the language design level to issues such as lexical composability and error-handling. In Section 4 we describe a high-level algorithm which implements local lexing as an extension of Earley's algorithm. We have formally verified the correctness of this algorithm in Isabelle/HOL [6], and provide an outline of this correctness proof in Section 5. The full proof (a total of 11411 lines or about 230 pages) is available in [12]. We also provide a practical library for local lexing [13], written in Scala/Scala.js [7, 8]. The library contains examples of its application to the examples in Section 3. Before concluding, we discuss further related work in Section 6.

2. Definition of Local Lexing

Before defining local lexing, we remind the reader of a few basic notions. For a set U we let U^* denote the set of sequences with elements in U , $\varepsilon \in U^*$ denotes the empty sequence, and for two sequences $\alpha \in U^*$ and $\beta \in U^*$ we let $\alpha\beta \in U^*$ denote their concatenation. Given a sequence $\alpha \in U^*$, we denote its length by $|\alpha|$ and let $\alpha_i \in U$ denote the i -th element of α for $i \in \{0, \dots, |\alpha| - 1\}$. A context-free grammar is a quadruple $(\mathfrak{N}, \mathfrak{T}, \mathfrak{R}, \mathfrak{S})$, where \mathfrak{N} is the set of nonterminals, \mathfrak{T} the set of terminals (such that \mathfrak{N} and \mathfrak{T} are disjoint), $\mathfrak{R} \subseteq \mathfrak{N} \times (\mathfrak{N} \cup \mathfrak{T})^*$ the rules of the grammar and $\mathfrak{S} \in \mathfrak{N}$ the start symbol. Instead of $(N, \alpha) \in \mathfrak{R}$ we often write $N \rightarrow \alpha$. For $\alpha, \beta \in (\mathfrak{N} \cup \mathfrak{T})^*$ we say $\alpha \Rightarrow \beta$ iff there are α_0, N, α_1 and γ such that $\alpha = \alpha_0 N \alpha_1$, $\beta = \alpha_0 \gamma \alpha_1$ and $N \rightarrow \gamma$. Furthermore, we write $\stackrel{*}{\Rightarrow}$ for the reflexive and transitive closure of \Rightarrow . We define the language \mathcal{L} of a grammar G as the set of all sequences of terminals derivable from the start symbol, $\mathcal{L} = \{w \in \mathfrak{T}^* \mid \mathfrak{S} \stackrel{*}{\Rightarrow} w\}$. Furthermore we define the set of *prefixes* $\mathcal{L}_{\text{prefix}}$ of G via

$$\mathcal{L}_{\text{prefix}} = \left\{ w \in \mathfrak{T}^* \mid \exists \alpha \in (\mathfrak{N} \cup \mathfrak{T})^*. \mathfrak{S} \stackrel{*}{\Rightarrow} w\alpha \right\}.$$

Definition 2.1 (Token). Given a set of terminals \mathfrak{T} , and a set of characters Σ , a *token* x is a pair $x = (t, c) \in \mathfrak{T} \times \Sigma^*$. In examples we will often use the notation

$$\frac{c}{t}$$

for the token x . We call the token *empty* iff $|c| = 0$. We define $[x] = t$ and $\bar{x} = c$. We lift these notations in the obvious manner to sequences of tokens: given a token sequence

$q = x_0 \dots x_r \in (\mathfrak{T} \times \Sigma^*)^*$ we define $[q] = [x_0] \dots [x_r] \in \mathfrak{T}^*$ and $\bar{q} = \bar{x}_0 \dots \bar{x}_r \in \Sigma^*$.

Definition 2.2 (Local Lexing). Given a set of terminals \mathfrak{T} , and a set of characters Σ , we call a pair (Lex, Sel) a *local lexing* (with respect to \mathfrak{T} and Σ) iff:

- The *lexer* Lex assigns to each terminal $t \in \mathfrak{T}$ a lexing function $\text{Lex}(t)$ which, given a sequence of characters $D \in \Sigma^*$ and a position $k \in \{0, \dots, |D|\}$, returns a set consisting of tokens (t, c) such that $k + |c| \leq |D|$ and $c_i = D_{k+i}$ for all i such that $0 \leq i \leq |c| - 1$.
- The *selector* Sel takes two token sets A and B such that $A \subseteq B$ and returns a token set $\text{Sel}(A, B)$ such that $A \subseteq \text{Sel}(A, B) \subseteq B$. We usually define Sel indirectly by defining a strict partial order $<_{\text{Sel}}$ on tokens and setting

$$\text{Sel}(A, B) = A \cup \{x \in B \mid \forall y \in B. \neg x <_{\text{Sel}} y\}.$$

Semantics of Local Lexing Given a local lexing $\ell\ell$, what is its semantics, i.e. how does it convert a sequence D of characters into a set $\ell\ell(D)$ of token sequences? Note that while for *defining* a particular local lexing $\ell\ell$ we do not need a context-free grammar, just its sets of terminals and characters, the *semantics* of $\ell\ell$ is dependent upon a grammar. In the following we will often call token sequences *paths*.

The conversion process works as follows: We go through D from left to right, producing sets of paths $\mathcal{P}_k^u \subseteq (\mathfrak{T} \times \Sigma^*)^*$ along the way. The index $k \in \{0, \dots, |D|\}$ denotes the position in D we are looking at, and we use the index $u \in \{0, 1, \dots\}$ to track iteratively generated $\mathcal{P}_k^0, \mathcal{P}_k^1, \dots$ at this position. Let's assume now that we have reached position k in D , having so far produced the set of paths \mathcal{P}_k^0 ; in case we are at the very beginning of D such that $k = 0$ we assume $\mathcal{P}_0^0 = \{\varepsilon\}$. We will have been careful to only produce $p \in \mathcal{P}_k^0$ such that $[p] \in \mathcal{L}_{\text{prefix}}$. Now, which token should we produce next? For sure, the token to produce next must be a member of the set

$$\mathcal{X}_k = \{x \in \mathfrak{T} \times \Sigma^* \mid x \in \text{Lex}([x])(D, k)\}$$

because according to Lex these are the only tokens which start at position k in D . Furthermore, we are only interested in the subset \mathcal{W}_k^0 of those members of \mathcal{X}_k which can continue a sequence in \mathcal{P}_k^0 to a sequence of terminals in $\mathcal{L}_{\text{prefix}}$:

$$\mathcal{W}_k^0 = \{x \in \mathcal{X}_k \mid \exists p \in \mathcal{P}_k^0. |\bar{p}| = k \wedge [px] \in \mathcal{L}_{\text{prefix}}\}.$$

The selector decides whether and how to constrain any remaining ambiguity in the token selection: The set

$$\mathcal{Z}_k^1 = \text{Sel}(\emptyset, \mathcal{W}_k^0)$$

contains those tokens we use to create the next set of paths \mathcal{P}_k^1 .

If \mathcal{Z}_k^1 does not contain any empty tokens, then \mathcal{P}_k^1 is just

$$\mathcal{P}_k^1 = \text{Append}_k \mathcal{Z}_k^1 \mathcal{P}_k^0,$$

where

$$\text{Append}_k T P = P \cup \{pt \mid p \in P \wedge |\bar{p}| = k \wedge t \in T \wedge [pt] \in \mathcal{L}_{\text{prefix}}\},$$

and because we only added paths q with $|\bar{q}| > k$ we can simply proceed to position $k + 1$ via $\mathcal{P}_{k+1}^0 = \mathcal{P}_k^1$.

But if \mathcal{Z}_k^1 does indeed contain empty tokens, then there might be newly added paths q with $|\bar{q}| = k$, and we might be able to extend these paths even further. Therefore we define \mathcal{P}_k^1 more generally as

$$\mathcal{P}_k^1 = \text{limit} (\text{Append}_k \mathcal{Z}_k^1) \mathcal{P}_k^0,$$

where

$$\text{limit } f X = \bigcup_{n=0}^{\infty} f^n(X).$$

The fact that now $\mathcal{P}_k^1 \setminus \mathcal{P}_k^0$ may contain paths q with $|\bar{q}| = k$ means that potentially more tokens in \mathcal{X}_k become eligible to extend paths which stop at position k . We therefore keep repeating the above procedure (potentially infinitely often) until we are sure to have produced all eligible tokens at position k by forming monotone chains

$$\begin{array}{ccccccc} \mathcal{W}_k^0 & \subseteq & \mathcal{W}_k^1 & \subseteq & \mathcal{W}_k^2 & \subseteq & \dots \\ \cup & & \cup & & \cup & & \\ \mathcal{Z}_k^0 & \subseteq & \mathcal{Z}_k^1 & \subseteq & \mathcal{Z}_k^2 & \subseteq & \dots \\ \mathcal{P}_k^0 & \subseteq & \mathcal{P}_k^1 & \subseteq & \mathcal{P}_k^2 & \subseteq & \dots \end{array}$$

where for $u \in \{0, 1, 2, \dots\}$ we recursively define

$$\begin{aligned} \mathcal{W}_k^u &= \{x \in \mathcal{X}_k \mid \exists p \in \mathcal{P}_k^u. |\bar{p}| = k \wedge [px] \in \mathcal{L}_{\text{prefix}}\} \\ \mathcal{Z}_k^{u+1} &= \text{Sel}(\mathcal{Z}_k^u, \mathcal{W}_k^{u+1}) \\ \mathcal{P}_k^{u+1} &= \text{limit} (\text{Append}_k \mathcal{Z}_k^{u+1}) \mathcal{P}_k^u. \end{aligned}$$

We finalize the generation of token sequences at position k by defining

$$\mathcal{P}_k^\infty = \bigcup_{u=0}^{\infty} \mathcal{P}_k^u$$

and move on to position $k + 1$ via $\mathcal{P}_{k+1}^0 = \mathcal{P}_k^\infty$. To complete the given set of recursive equations, we define $\mathcal{Z}_k^0 = \emptyset$ for all positions k .

Once we have arrived at the end of the character sequence D , we finish the conversion of D into the set $\ell\ell(D)$ of token sequences via defining $\mathfrak{P} = \mathcal{P}_{|D|}^\infty$ and

$$\ell\ell(D) = \{p \in \mathfrak{P} \mid |\bar{p}| = |D| \wedge [p] \in \mathcal{L}\}.$$

Note that the local lexing semantics does not depend on the particular form of the grammar, but only on \mathcal{L} and $\mathcal{L}_{\text{prefix}}$. If the grammar contains no unproductive nonterminals, i.e. if for all $X \in \mathfrak{N}$ there is $w \in \Sigma^*$ such that $X \xRightarrow{*} w$, then $\mathcal{L}_{\text{prefix}}$ is uniquely determined by \mathcal{L} and thus in this case the local lexing semantics only depends on \mathcal{L} alone.

Let us also comment on why the selector takes two arguments. It is tempting (and indeed this was the first thing we tried) to let the selector act only on a single argument, as in $\mathcal{Z}_k^{u+1} = \text{Sel } \mathcal{W}_k^{u+1}$, but this destroys the property that the \mathcal{Z}_k^u form a monotone chain, which turned out to be important for the correctness of our algorithm for local lexing (see Section 5).

3. Applications of Local Lexing

In this section we explore local lexing and its applications through a range of examples.

Example 3.1 (Traditional Lexical Specifications). Turning a traditional lexical specification into the definition of a local lexing is straightforward. Let the traditional specification be defined over an input alphabet Σ and given by n pairs

$$(r_1, t_1) \dots (r_n, t_n).$$

The r_i are regular expressions over Σ , none of which match $\varepsilon \in \Sigma^*$, and the set of terminals \mathfrak{T} consists of n different terminals t_1, \dots, t_n . For each $t_i \in \mathfrak{T}$ we then define

$$\text{Lex}(t_i)(D, k) = \{(t_i, c)\},$$

where c is the longest prefix of $D_k \dots D_{|D|-1}$ such that r_i matches c . If r_i matches no prefix of $D_k \dots D_{|D|-1}$ then we define $\text{Lex}(t_i)(D, k) = \emptyset$.

We define a strict partial order $<_{\text{Sel}}$ on the set of tokens which encodes the longest-match and priority rules:

$$(t_i, c) <_{\text{Sel}} (t_j, d) \quad \text{iff} \quad |c| < |d| \vee (|c| = |d| \wedge i < j).$$

This implies that $\text{Sel}(\emptyset, X)$ will be either empty or a singleton because $<_{\text{Sel}}$ is total on all possible X .

To endow the local lexing $\ell\ell = (\text{Lex}, \text{Sel})$ with the equivalent semantics to traditional lexing, we define the context-free grammar $G = (\{S, T\}, \mathfrak{T}, \mathfrak{R}, S)$ where

$$\mathfrak{R} = \left\{ \begin{array}{lcl} S & \rightarrow & ST, \\ S & \rightarrow & \varepsilon, \\ T & \rightarrow & t_1, \\ & \vdots & \\ T & \rightarrow & t_n \end{array} \right\}.$$

Because of $\mathcal{L}_{\text{prefix}} = \mathcal{L} = \mathfrak{T}^*$ this grammar poses no additional constraints on the local lexing process, and therefore $\ell\ell(D) = \emptyset$ iff traditional lexing of D would lead to an error, and $\ell\ell(D) = \{p\}$ iff traditional lexing of D would yield the token sequence p .

Example 3.2 (Infinite Set of Token Sequences). We change the previous example slightly and allow the r_i to also match the empty character sequence ε . As a concrete example consider $\Sigma = \{a\}$, $n = 1$ and let r_1 be a regular expression which matches any (possibly empty) sequence of a 's. Then

using the same grammar G as in the previous example, we obtain for example

$$\ell\ell(aa) = \left\{ \frac{aa}{t_1}, \frac{aa \ \varepsilon}{t_1 \ t_1}, \frac{aa \ \varepsilon \ \varepsilon}{t_1 \ t_1 \ t_1}, \frac{aa \ \varepsilon \ \varepsilon \ \varepsilon}{t_1 \ t_1 \ t_1 \ t_1}, \dots \right\},$$

which is an infinite set.

Because local lexing intertwines lexing and parsing, determining for a character sequence $D \in \Sigma^*$ all possible token sequences $\ell\ell(D)$ involves finding valid parses of $[p]$ for all $p \in \ell\ell(D)$. The following examples demonstrate this interplay between lexing and parsing and the role of the selector Sel in it.

Example 3.3. Consider the grammar $H = (\mathfrak{N}, \mathfrak{T}, \mathfrak{R}, \mathfrak{S})$ where $\mathfrak{N} = \{S, A, E\}$, $\mathfrak{T} = \{plus, minus, id, symbol\}$, $\mathfrak{R} = \{S \rightarrow S \ plus \ A, S \rightarrow S \ minus \ A, S \rightarrow A, A \rightarrow A \ E, A \rightarrow E, E \rightarrow id, E \rightarrow symbol\}$, $\mathfrak{S} = S$. The input characters are $\Sigma = \{+, -, a, b, c\}$. As previously, we specify Lex simply by associating the terminals with regular expressions: The terminal *plus* recognizes the character $+$, *minus* recognizes the character $-$, *id* recognizes any nonempty sequence of letters and *symbol* recognizes any nonempty sequence of letters and hyphens $-$. Let D denote the character sequence $a-b+c$.

We choose the simplest option for $<_{Sel}$ and define

$$<_{Sel} = \emptyset,$$

i.e. no token has a higher priority than another token. This implies $Sel(A, B) = B$ for all token sets $A \subseteq B$. The result of determining $\ell\ell(D)$ is shown in Figure 1. Because *symbol* overlaps with both *identifier* and *minus*, D can be interpreted in eight different ways as a token sequence. Because of the longest-match rule for regular expressions,

$$\frac{a}{symbol} \ \frac{-}{minus} \ \frac{b}{id} \ \frac{+}{plus} \ \frac{c}{id} \notin \ell\ell(D),$$

excluded together with several other token sequences which would otherwise qualify.

For all $p \in \ell\ell(D)$ the terminal sequence $[p]$ yields a valid parse tree, i.e. there are 8 different ways to parse D although H by itself is an unambiguous grammar; all of them are jointly depicted in the parse graph in Figure 2. Ambiguities are depicted as dashed lines in the graph: whenever there are multiple ways to proceed with the derivation of a nonterminal, each alternative is connected with the nonterminal by a dashed line.

Example 3.4. We use the same grammar H and the same lexer Lex as in Example 3.3, but choose a selector Sel such that *symbol* has lower priority than *id* and *minus*:

$$x <_{Sel} y \quad \text{iff} \quad [x] = symbol \wedge [y] \in \{id, minus\}.$$

This results in a unique lexing of $a-b+c$,

$$\ell\ell(a-b+c) = \left\{ \frac{a}{id} \ \frac{-}{minus} \ \frac{b}{id} \ \frac{+}{plus} \ \frac{c}{id} \right\},$$

$$\ell\ell(a-b+c) = \left\{ \begin{array}{l} \frac{a}{id} \ \frac{-}{minus} \ \frac{b}{id} \ \frac{+}{plus} \ \frac{c}{id} , \\ \frac{a}{id} \ \frac{-}{minus} \ \frac{b}{id} \ \frac{+}{plus} \ \frac{c}{symbol} , \\ \frac{a}{id} \ \frac{-}{minus} \ \frac{symbol}{symbol} \ \frac{plus}{plus} \ \frac{id}{id} , \\ \frac{a}{id} \ \frac{-}{minus} \ \frac{symbol}{symbol} \ \frac{plus}{plus} \ \frac{symbol}{symbol} , \\ \frac{a}{id} \ \frac{-}{minus} \ \frac{symbol}{symbol} \ \frac{plus}{plus} \ \frac{symbol}{symbol} , \\ \frac{a-b}{symbol} \ \frac{+}{plus} \ \frac{c}{id} , \\ \frac{a-b}{symbol} \ \frac{+}{plus} \ \frac{c}{symbol} , \\ \frac{a-b}{symbol} \ \frac{+}{plus} \ \frac{c}{symbol} \end{array} \right\}$$

Figure 1. Lexing $a-b+c$ in Example 3.3

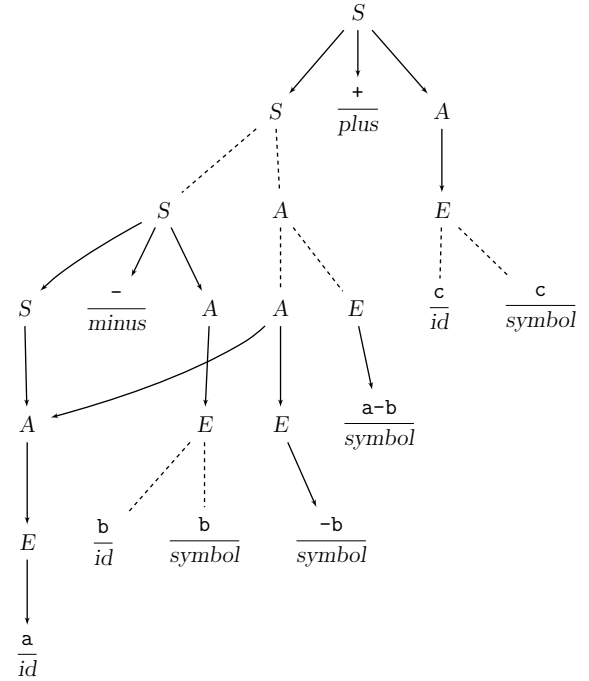


Figure 2. Parsing $a-b+c$ in Example 3.3

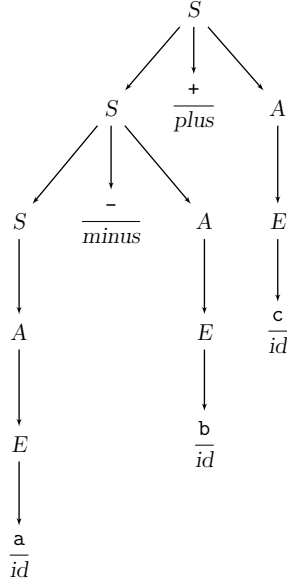


Figure 3. Parsing $a-b+c$ in Example 3.4

and thus also in a unique parsing as shown in Figure 3.

Example 3.5. We again leave H and Lex fixed. This time we define $<_{Sel}$ such that longer tokens have higher priority:

$$x <_{Sel} y \quad \text{iff} \quad |x| < |y|.$$

This yields two possible lexings for $a-b+c$,

$$\ell\ell(a-b+c) = \left\{ \begin{array}{l} \frac{a-b}{symbol} \quad \frac{+}{plus} \quad \frac{c}{id}, \\ \frac{a-b}{symbol} \quad \frac{+}{plus} \quad \frac{c}{symbol} \end{array} \right\},$$

and leads to the ambiguous parsing shown on the left hand side of Figure 4.

Example 3.6. We choose to modify the previous example such that its remaining ambiguity is resolved in favour of id by defining $x <_{Sel} y$ iff

$$|x| < |y| \vee (|x| = |y| \wedge [x] = symbol \wedge [y] = id).$$

This leads to a unique lexing for $a-b+c$,

$$\ell\ell(a-b+c) = \left\{ \frac{a-b}{symbol} \quad \frac{+}{plus} \quad \frac{c}{id} \right\},$$

and yields the unique parsing depicted on the right hand side of Figure 4.

Example 3.7 (The Lexer Hack). Consider the expression $(a) * b$ given in the programming language C. The meaning of this expression depends on whether a is a type identifier or a variable identifier. If it is a type identifier, then the expression is to be interpreted as a type cast, otherwise as

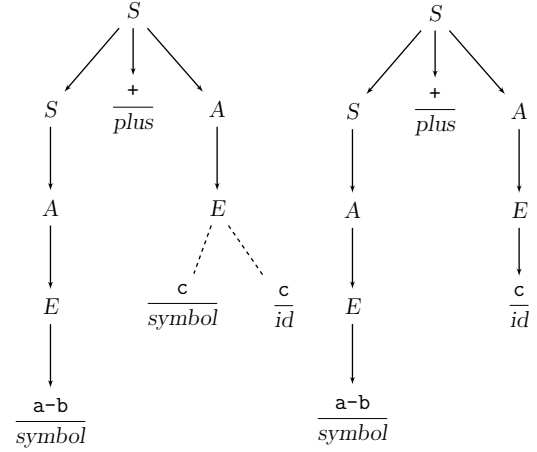


Figure 4. Parsing $a-b+c$ in Ex. 3.5 (left) and Ex. 3.6 (right)

a multiplication. Because type identifiers cannot be distinguished from variable identifiers by their look, in traditional parsing a problem arises, because the lexer phase is supposed to happen *before* the parsing phase and any semantic analysis, but the terminal type of a really depends on information from the semantic analysis. The traditional solution, to manually direct feedback from the semantic analysis back into the lexer, is known as *the lexer hack* [4]. With local lexing, the lexer / parser can offer *both* alternatives, and let later stages of the analysis pick the right one, thus decoupling parsing and semantic analysis. The grammar C demonstrates this, where $\mathfrak{T} = \{typeid, id, asterisk, left, right\}$ and

$$\mathfrak{R} = \left\{ \begin{array}{l} Expr \rightarrow Mul, \\ Expr \rightarrow Cast, \\ Expr \rightarrow Deref, \\ Expr \rightarrow id, \\ Expr \rightarrow left\ Expr\ right, \\ Mul \rightarrow Expr\ asterisk\ Expr, \\ Cast \rightarrow left\ Type\ right\ Expr, \\ Deref \rightarrow asterisk\ Expr, \\ Type \rightarrow typeid \end{array} \right\}.$$

The lexer Lex is chosen in the obvious way, together with an empty $<_{Sel}$. The resulting ambiguous parse graph for $(a) * b$ is shown in Figure 5, and

$$\ell\ell((a) * b) = \left\{ \begin{array}{l} \frac{(\quad a \quad)}{left\ id\ right} \quad \frac{*}{asterisk} \quad \frac{b}{id}, \\ \frac{(\quad a \quad)}{left\ typeid\ right} \quad \frac{*}{asterisk} \quad \frac{b}{id} \end{array} \right\}.$$

Schrödinger's Token Example 3.7 is a special case of a scenario where the conversion from character sequences to token sequences is ambiguous, but where the token *boundaries* are always the same in all alternative token sequences. The tokens appearing in such a scenario have been christened Schrödinger's tokens [5]. Local lexing is more power-

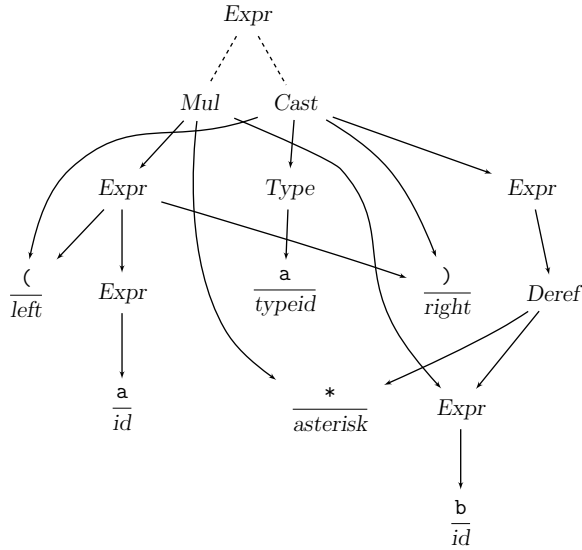


Figure 5. Parsing (a) $\ast b$ in C

ful than the Schrödinger's token approach and (at least conceptually) subsumes it.

Ruby Slippers The Marpa parser [10] is an Earley-based parsing library which advocates the use of a technique called *Ruby Slippers* [11]. This technique takes advantage of the fact that the Earley parser “knows” which tokens it expects at any given stage of the parse progress. Marpa has an interface through which the parser can communicate with the scanner to negotiate which token to scan next, thus allowing for sophisticated error handling.

Ruby Slippers and local lexing are both children of the same insight, namely that the scanning and parsing stages should communicate because the parser has information about which tokens it expects next. Local lexing though takes this insight to a new level which in principle is independent from a particular parsing algorithm like Earley. In this sense, local lexing can be seen as providing a rigorous semantics for certain uses of the Ruby Slippers technique.

The next example demonstrates how local lexing can be used to specify lightweight error recovery as part of the language design.

Example 3.8 (Error Recovery). Consider a grammar for simple arithmetic expressions, where the nonterminals are given by $\{Expr, Sum, Mul, Atom\}$, the terminals by

$\{plus, mul, id, num, left, right\}$, and the rules by

<i>Expr</i>	\rightarrow	<i>Sum</i> ,
<i>Sum</i>	\rightarrow	<i>Sum plus Mul</i> ,
<i>Sum</i>	\rightarrow	<i>Mul</i> ,
<i>Mul</i>	\rightarrow	<i>Mul mul Atom</i> ,
<i>Mul</i>	\rightarrow	<i>Atom</i> ,
<i>Atom</i>	\rightarrow	<i>left Sum right</i> ,
<i>Atom</i>	\rightarrow	<i>id</i> ,
<i>Atom</i>	\rightarrow	<i>num</i> .

The input characters are $\Sigma = \{+, *, (,), 0 \dots 9, a \dots z\}$, and *Lex* is specified such that *plus* recognizes the character +, *mul* recognizes the character *, *left* recognizes the opening bracket (, *right* recognizes the closing bracket), *id* recognizes any nonempty sequence of letters and digits starting with a letter, and *num* recognizes any nonempty sequence of digits. Consider now the following string *w* which is invalid with respect to the language we just specified:

$$2(a^* +) + (1$$

Instead of simply diagnosing that there is some parse error at position $k = 1$, we would like to recover some of the structure of w , for example for providing better error messages or for a rich interactive editing experience. To simplify this task, we first make our grammar more permissive by adding the rule $Mul \rightarrow Mul\ Atom$. This has the effect that a string like $2 \times$ becomes legal, representing the multiplication of 2 and \times . Such notation is common mathematical practice and thus seems like a justifiable design choice. We then introduce three new terminals: *e-atom* and *e-right* both recognize the empty string ε only, and *e-superfluous* recognizes the closing bracket $)$. We incorporate these new terminals into the grammar by adding the following rules:

$$\begin{array}{ll} \text{Mul} & \rightarrow \text{Mul e-superfluous,} \\ \text{Atom} & \rightarrow \text{left Sum e-right,} \\ \text{Atom} & \rightarrow \text{e-atom} \end{array}$$

Finally, we choose the selector Sel such that the three added error terminals have *lower priority than all other terminals*. Paths in $\ell\ell(D)$ that contain error terminals will only be considered by us if $\ell\ell(D)$ contains no paths without error terminals. Figure 6 shows the result of applying the updated grammar to w . The corresponding path is

$$\frac{\frac{2}{num} \left(\frac{a}{left} * \frac{\epsilon}{id} + \frac{\epsilon}{mul} \right)}{e-atom} \frac{plus}{e-atom} \frac{\epsilon}{right} \\ \frac{)}{e-superfluous} + \left(\frac{1}{plus} \frac{\epsilon}{left} \frac{\epsilon}{num} \frac{\epsilon}{e-right} \right).$$

Testing convinces us that the updated grammar can indeed successfully parse all $D \in \Sigma^*$, and does so unambiguously.

Whitespace In our examples we have avoided to make use of whitespace. In traditional parsing there are essentially two different ways of dealing with whitespace:

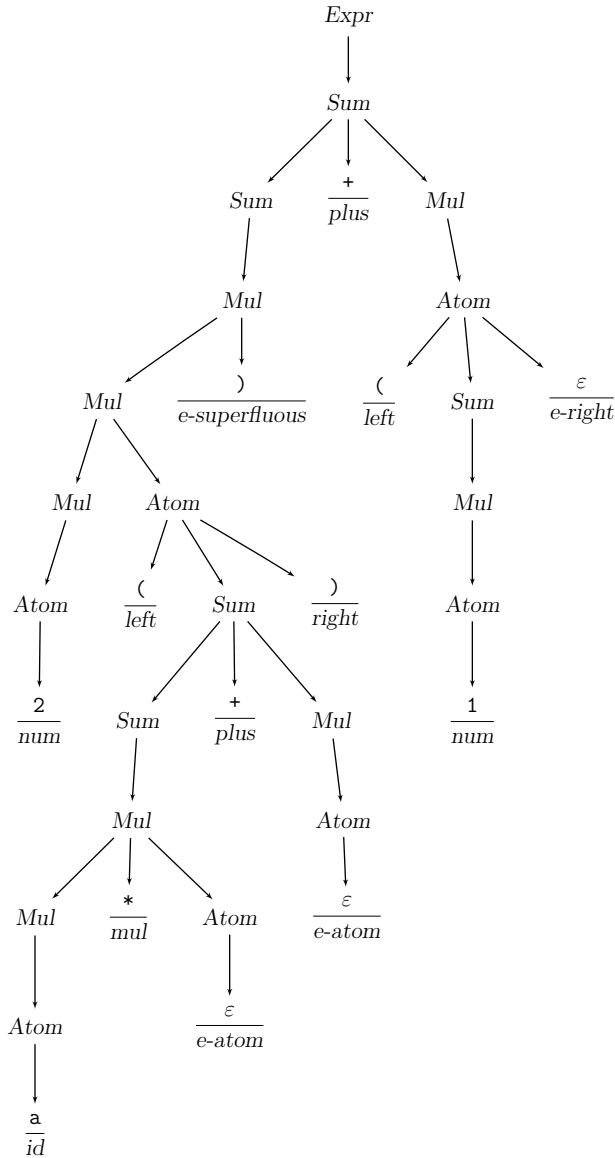


Figure 6. Parsing $2(a^{*+}) + (1$ in Example 3.8

1. One approach is to explicitly incorporate whitespace terminals into the grammar rules. This can become cumbersome and error-prone if done manually, because usually whitespace can legally appear almost everywhere.
2. The other approach is to handle whitespace at the lexical stage exclusively.

The first approach applies to local lexing as well. The second approach does not directly apply, as there is no single lexical stage anymore with local lexing. Nevertheless, it seems that local lexing allows to combine the convenience of the second approach with the fine-grained control of the first one by using extended attribute grammars to specify layout constraints, making it possible to tackle layout-sensitive

languages. The combination of local lexing with layout-sensitivity is work in progress and beyond the scope of this paper.

4. Implementing Local Lexing

Given a grammar G and a local lexing $\ell\ell$, let us define the *character language* \mathcal{L}_Σ of G and $\ell\ell$ by

$$\mathcal{L}_\Sigma = \{D \in \Sigma^* \mid \ell\ell(D) \neq \emptyset\}.$$

How do we build a recognizer for \mathcal{L}_Σ ?

Lexing Driving Parsing Our first attempt might be to directly apply the semantics of local lexing after having picked a parsing algorithm which is capable of recognizing both the prefixes $\mathcal{L}_{\text{prefix}}$ and the language \mathcal{L} of G . Most of the popular parsing algorithms would be suitable for this, such as LL, LR or Earley parsing.

While this approach will work in many cases, it is inefficient to treat the parsing algorithm as a black box which is repeatedly asked whether a given sequence of terminals is in $\mathcal{L}_{\text{prefix}}$ or not. More importantly, as Example 3.2 shows there are finite grammars which nevertheless produce infinite sets of token sequences and for which this approach would therefore fail by getting stuck in a non-terminating path generating loop.

Parsing Driving Lexing A better approach seems to be to inverse above approach and to let the parsing progress drive the lexing process. Often it will be possible to predict from the internal parser state which terminal is expected next. For this to work, we need to modify the parsing algorithm so that it not only knows about terminals \mathcal{T} , but also about characters Σ . The Earley algorithm seems to be best suited to be adapted to such a purpose, as it works on all context-free grammars, copes gracefully with ambiguity, and is easily extended with top-down, left-right, and bottom-up parametricity. This is why Earley-based parsing is our main focus here. Nevertheless, studying how to modify other parsing algorithms for local lexing is interesting and of potentially great practical interest as well; experiments indicate that in particular the LR(1) parsing algorithm can be modified to facilitate local lexing in a natural and simple way.

Earley’s Algorithm We first describe (a high-level version of) Earley’s original algorithm, assuming $\Sigma = \mathfrak{T}$. To recognize an input $D \in \Sigma^*$ as belonging to \mathcal{L} , it computes *items*; an item is a quadruple (r, d, i, j) where $r = (N \rightarrow \alpha\beta) \in \mathfrak{R}$ is a rule of the grammar, $d = |\alpha|$ is a position within that rule demarking the current parsing progress, $i \in \{0, \dots, |D|\}$ is the origin of the item, and $j \in \{i, \dots, |D|\}$ is the bin of the item. An alternative way to write the item is as

$$(N \rightarrow \alpha \bullet \beta, i, j).$$

Earley's algorithm builds a monotone chain of item sets

$$\mathcal{I}_0 \subseteq \mathcal{I}_1 \subseteq \mathcal{I}_2 \subseteq \dots \subseteq \mathcal{I}_{|D|} = \mathfrak{I}.$$

$$\begin{aligned}
\text{Init} &= \{(\mathfrak{S} \rightarrow \bullet \alpha, 0, 0) \mid \mathfrak{S} \rightarrow \alpha \in \mathfrak{R}\} \\
\text{Predict } k \ I &= I \cup \\
&\quad \{(M \rightarrow \bullet \gamma, k, k) \mid \exists N \alpha \beta i. \\
&\quad (N \rightarrow \alpha \bullet M \beta, i, k) \in I \wedge (M \rightarrow \gamma) \in \mathfrak{R}\} \\
\text{Complete } k \ I &= I \cup \\
&\quad \{(N \rightarrow \alpha M \bullet \beta, i, k) \mid \exists j \gamma. \\
&\quad (N \rightarrow \alpha \bullet M \beta, i, j) \in I \wedge (M \rightarrow \gamma \bullet, j, k) \in I\} \\
\text{Scan } k \ I &= I \cup \\
&\quad \{(N \rightarrow \alpha X \bullet \beta, i, k+1) \mid k < |D| \wedge X = D_k \wedge \\
&\quad (N \rightarrow \alpha \bullet X \beta, i, k) \in I\}
\end{aligned}$$

Figure 7. Building Blocks of Earley's Algorithm

To this end, we define an initial item set Init and monotone operators Predict , Complete and Scan which all take a position $k \in \{0, \dots, |D|\}$ and an item set I and return an augmented item set (Figure 7). We then define for $k \in \{0, \dots, |D|\}$ the operator π_k by

$$\pi_k I = \text{limit } (\text{Scan } k \circ \text{Complete } k \circ \text{Predict } k) I,$$

reusing the limit operator introduced in Section 2, and use π_k to recursively define the sets \mathcal{I}_k by

$$\begin{aligned}
\mathcal{I}_0 &= \pi_0 \text{Init}, \\
\mathcal{I}_k &= \pi_k \mathcal{I}_{k-1} \text{ for } k > 0.
\end{aligned}$$

Theorem 4.1 (Correctness of Earley's Algorithm). *Earley's algorithm is both sound and complete, i.e.*

$$\mathfrak{S} \xRightarrow{*} D \quad \text{iff} \quad \exists \alpha. (\mathfrak{S} \rightarrow \alpha \bullet, 0, |D|) \in \mathfrak{I}.$$

Proof. This is covered by Theorem 4.2 for the special case

$$\begin{aligned}
\Sigma &= \mathfrak{T}, \\
\text{Lex}(t)(D, k) &= \begin{cases} \{(t, t)\} & \text{for } k < |D| \wedge D_k = t \\ \emptyset & \text{otherwise} \end{cases}, \\
<_{\text{Sel}} &= \emptyset.
\end{aligned}$$

□

Earley's Algorithm with Local Lexing We now assume that we have a local lexing $\ell\ell$ and thus Σ and \mathfrak{T} do not necessarily coincide anymore. We leave Init , Predict and Complete unchanged, but we need an updated Scan operator that works on tokens instead of characters, and a new operator Tokens (Figure 8).

The operation $\text{Tokens } T \ k \ I$ first determines all the candidate terminals that could possibly appear next at position k

$$\begin{aligned}
\text{Tokens } T \ k \ I &= \\
\text{Sel } T \ \{x \mid \exists X \ N \ \alpha \ \beta \ i. X \in \mathfrak{T} \wedge \\
&\quad (N \rightarrow \alpha \bullet X \beta, i, k) \in I \wedge x \in \text{Lex}(X)(D, k)\} \\
\text{Scan } T \ k \ I &= I \cup \\
&\quad \{(N \rightarrow \alpha X \bullet \beta, i, k + |c|) \mid (X, c) \in T \wedge \\
&\quad (N \rightarrow \alpha \bullet X \beta, i, k) \in I\}
\end{aligned}$$

Figure 8. A New Scanner

in D according to I . It then determines which of those candidate terminals can actually be lexed as tokens at that position. It applies the selector Sel to it and returns the resulting set of tokens. Finally, the Scan operator is easily adapted to work on tokens instead of single characters. We iteratively compute the sets $\mathcal{T}_k^0, \mathcal{T}_k^1, \dots$ of tokens at position k , and these act as arguments T to both Tokens and Scan . Accordingly, we need to update the definition of π_k to take the additional argument T into account:

$$\pi_k T \ I = \text{limit } (\text{Scan } T \ k \circ \text{Complete } k \circ \text{Predict } k) I.$$

Furthermore, it is now possible that scanning at position k might add new items to bin k due to the existence of empty tokens, therefore enlarging the set of eligible terminals at position k . To cope with this we keep applying the operator π_k with updated token sets until it converges:

$$\begin{aligned}
\mathcal{J}_0^0 &= \pi_0 \emptyset \text{Init} \\
\mathcal{J}_k^{u+1} &= \pi_k \mathcal{T}_k^{u+1} \mathcal{J}_k^u \\
\mathcal{I}_k &= \bigcup_{u=0}^{\infty} \mathcal{J}_k^u \\
\mathcal{J}_{k+1}^0 &= \pi_{k+1} \emptyset \mathcal{I}_k \\
\mathcal{T}_k^0 &= \emptyset \\
\mathcal{T}_k^{u+1} &= \text{Tokens } \mathcal{T}_k^u \ k \ \mathcal{J}_k^u.
\end{aligned}$$

Note that in case of $\mathcal{J}_k^{u+1} = \mathcal{J}_k^u$ we have $\mathcal{I}_k = \mathcal{J}_k^u$, thus the computation of \mathcal{I}_k can stop at that point. In particular, if \mathcal{T}_k^1 does not contain any empty tokens, then the computation of \mathcal{I}_k simplifies to $\mathcal{I}_k = \mathcal{J}_k^1$.

Above equations for computing $\mathfrak{I} = \mathcal{I}_{|D|}$ show an obvious correspondence to the equations we used for defining \mathfrak{P} in Section 2. And indeed, Earley's algorithm with local lexing is correct with respect to the local lexing semantics:

Theorem 4.2 (Correctness of Earley's Algorithm with Local Lexing). *Earley's algorithm with local lexing is both sound and complete, i.e.*

$$D \in \mathcal{L}_\Sigma \quad \text{iff} \quad \exists \alpha. (\mathfrak{S} \rightarrow \alpha \bullet, 0, |D|) \in \mathfrak{I}.$$

Proof. See Section 5. □

The above correctness result comes with a caveat: It may be the case that the computation requires an infinite amount of time and space. If the grammar is finite though, then for an input $D \in \Sigma^*$ the size of \mathcal{J} is bounded by

$$\left(\binom{|D|+1}{2} + |D| + 1 \right) \sum_{N \rightarrow \alpha \in \mathfrak{R}} 1 + |\alpha|,$$

and thus the computation will require only a finite amount of time and space (assuming *Sel* and *Lex* require only a finite amount in the first place).

Practical Implementation We have developed a practical library for local lexing written in Scala/Scala.js which can be used to try out the examples in Section 3 [13]. It is a fairly naive proof-of-concept implementation and not optimized for data structures at all. For future versions of the library we plan to examine which of the many established ideas for making Earley parsing faster also apply (at least partially) to our case.

5. Proof of Theorem 4.2

In this section we give a short outline of the proof of Theorem 4.2. The full formal proof is available as Isabelle/HOL 2016 theory files and has been fully machine-checked for correctness [12]. To convince yourself that the formal proof really proves Theorem 4.2 we recommend first studying theories *CFG*, *LocalLexing* and *LLEarleyParsing*. These contain the basic definitions in (almost) the same notation as presented in this paper. You should then proceed to look at theory *MainTheorems*. It contains the theorem *Correctness* which is the machine-checked counterpart of Theorem 4.2.

Proof Outline There is an intuitive correspondence between the sets \mathcal{P}_k^u and the sets \mathcal{J}_k^u . Clarifying this correspondence is the most important step towards proving the correctness of the algorithm.

Definition 5.1 (Valid and Generated Items). We call an item

$$(N \rightarrow \alpha \bullet \beta, i, j) \in \mathcal{J}$$

p-valid for some token sequence $p \in \mathfrak{P}$ iff there is $u \in \{0, \dots, |p|\}$ such that

$$\begin{aligned} |\bar{p}| &= j, \\ |\overline{p_0 \dots p_{u-1}}| &= i, \\ \mathfrak{S} &\stackrel{*}{\Rightarrow} [p_0 \dots p_{u-1}] N \gamma \quad \text{for some } \gamma, \\ \alpha &\stackrel{*}{\Rightarrow} [p_u \dots p_{|p|-1}]. \end{aligned}$$

For $P \subseteq \mathfrak{P}$ we say that P generates $\langle P \rangle$, where

$$\langle P \rangle = \{x \in \mathcal{J} \mid \exists p \in P. x \text{ is } p\text{-valid}\}.$$

This notion of validity has been inspired by the one introduced in [9] where it has been defined as an absolute property of an item. To make it work in our context, we had to define it not absolutely, but relatively with respect to a path.

The bulk of the proof consists then in proving $\mathcal{J} = \langle \mathfrak{P} \rangle$. We will not delve into the rather technical proof of this here, but we want to point out the following supporting theorem about paths:

Theorem 5.1. For all inputs D , for all $k \in \{0, \dots, |D|\}$ and for all $u \in \{0, 1, 2, \dots\}$ the following holds: Given $p, q \in \mathcal{P}_k^u$ such that $|\bar{p}| = |\overline{q_0 \dots q_{n-1}}| \leq k$ for some $n \in \{0, \dots, |q|\}$ and $[p q_n \dots q_{|q|-1}] \in \mathcal{L}_{\text{prefix}}$, it follows that $p q_n \dots q_{|q|-1} \in \mathcal{P}_k^u$.

Intuitively this means that when there are two paths $p \in \mathfrak{P}$ and $q \in \mathfrak{P}$ which meet at some position k , i.e. $p = ab$ and $q = cd$ with $|\bar{a}| = |\bar{c}| = k$, then they can crossover, i.e. both ad and cb will be in \mathfrak{P} as long as this makes sense with respect to the grammar. But p and q are not guaranteed to arrive at the same time u at position k , and so it might be that tokens that were around when p arrived are not there anymore when q arrives, and vice versa. The fact that the token sets \mathcal{Z}_k^u form a monotone chain at position k means that this cannot happen.

The semantics of local lexing is defined by mutually recursive equations which intertwine lexing and parsing, but once all token sets $\mathcal{Z}_k^\infty = \bigcup_{u=0}^\infty \mathcal{Z}_k^u$ have been established it is possible to disentangle lexical and grammatical matters again as another supporting theorem about paths shows:

Theorem 5.2. Let p be a sequence of tokens. Then $p \in \mathfrak{P}$ iff

$$a) \quad \forall_{0 \leq i < |p|} p_i \in \mathcal{Z}_{|\overline{p_0 \dots p_{i-1}}|}^\infty \quad \text{and} \quad b) \quad [p] \in \mathcal{L}_{\text{prefix}}.$$

From $\mathcal{J} = \langle \mathfrak{P} \rangle$ and with the help of Theorem 5.2 it is then straightforward to prove Theorem 4.2:

Proof. Let us first assume $D \in \mathcal{L}_\Sigma$. This implies $\ell\ell(D) \neq \emptyset$, which implies that there is a $p \in \mathfrak{P}$ with $|\bar{p}| = |D|$ and $\mathfrak{S} \stackrel{*}{\Rightarrow} [p]$. This means there is an α such that $\mathfrak{S} \rightarrow \alpha$ and $\alpha \stackrel{*}{\Rightarrow} [p]$. Therefore, $(\mathfrak{S} \rightarrow \alpha \bullet, 0, |D|)$ is *p*-valid and thus

$$(\mathfrak{S} \rightarrow \alpha \bullet, 0, |D|) \in \langle \mathfrak{P} \rangle = \mathcal{J}.$$

On the other hand, let us assume that there is an α with above property. This means that $(\mathfrak{S} \rightarrow \alpha \bullet, 0, |D|)$ is *p*-valid for some $p \in \mathfrak{P}$, i.e. there is $u \in \{0, \dots, |p|\}$ with

$$\begin{aligned} |\bar{p}| &= |D|, \\ |\overline{p_0 \dots p_{u-1}}| &= 0, \\ \mathfrak{S} &\stackrel{*}{\Rightarrow} [p_0 \dots p_{u-1}] \mathfrak{S} \gamma \quad \text{for some } \gamma, \\ \alpha &\stackrel{*}{\Rightarrow} [p_u \dots p_{|p|-1}]. \end{aligned}$$

Above facts together with Theorem 5.2 show that by dropping the first u empty tokens from p we obtain a path

$$p_u \dots p_{|p|-1} \in \ell\ell(D).$$

□

6. Further Related Work

A strong influence on our work has been the idea of *blackbox* Earley parsing presented in [1]. A blackbox is a (possibly third-party) parser component plugged into the Earley parsing framework by associating the blackbox with a nonterminal. Our *Lex* component can basically be viewed as a collection of blackboxes, but instead of associating them with nonterminals, we associate them with terminals. This makes it possible to treat blackboxes as a concept that is in principle independent from Earley parsing. Unlike the original work on blackboxes we also provide a method for disambiguation, via the selector *Sel*.

7. Conclusion

With hindsight local lexing is a simple concept, but it has taken us over two years to arrive at the concept as it is presented in this paper. Our algorithm for local lexing and the semantics of local lexing developed side-by-side during this time. There have been enough missteps along the way to finally make us formally verify our algorithm. Despite its simplicity, the examples from Section 3 show that local lexing is a versatile and unifying concept for designing syntax. We hope that you may find it useful, too.

References

- [1] Trevor Jim, Yitzhak Mandelbaum, David Walker. *Semantics and Algorithms for Data-dependent Grammars*, POPL 2010.
- [2] Brian Ford. *Parsing Expression Grammars: A Recognition Based Syntactic Foundation*, POPL 2004.
- [3] Eelco Visser. *Scannerless Generalized-LR Parsing*, Report P9707, University of Amsterdam 1997.
- [4] The lexer hack,
https://en.wikipedia.org/wiki/The_lexer_hack.
- [5] John Aycock, R. Nigel Horspool. *Schrödinger's token*, Software Practice and Experience, 2001.
- [6] Isabelle, <http://isabelle.in.tum.de>.
- [7] Scala, <http://scala-lang.org>.
- [8] Sébastien Doeraene. *Scala.js*, <http://scala-js.org>.
- [9] Cliff B. Jones. *Formal Development Of Correct Algorithms: An Example Based On Earley's Recogniser*, Proving Assertions about Programs, 1972.
- [10] Jeffrey Kegler. *The Marpa parser*, <http://jeffreykegler.github.io/Marpa-web-site/>.
- [11] Jeffrey Kegler. *Marpa and the Ruby Slippers*, http://blogs.perl.org/users/jeffrey_kegler/2011/11/marpa-and-the-ruby-slippers.html.
- [12] Steven Obua. *Isabelle Theories for Local Lexing (v1.1.0)*, doi: [10.5281/zenodo.322419](https://doi.org/10.5281/zenodo.322419).
- [13] Steven Obua. *Local Lexing Prototype Implementation (v1.0.1)*, doi: [10.5281/zenodo.322417](https://doi.org/10.5281/zenodo.322417).